Building Safe PaaS Clouds: a Survey on Security in Multitenant Software Platforms

Luis Rodero-Merino, Luis Vaquero, Eddy Caron, Frédéric Desprez, Adrian Muresan

Building Safe PaaS Clouds: a Survey on Security in Multitenant Software Platforms

Luis Rodero-Merino, Luis M. Vaquero, Eddy Caron, Frédéric Desprez, Adrian Muresan

UMR CNRS - ENS de Lyon - INRIA - UCB Lyon 5668, 46 allée d'Italie, F-69364 Lyon, France

Universidad Politécnica de Madrid, Facultad de Informática, B2 L3201, 28660 Boadilla del Monte, Spain

Hewlett-Packard Labs, Stoke Gufford BS34 8QZ, Bristol, UK

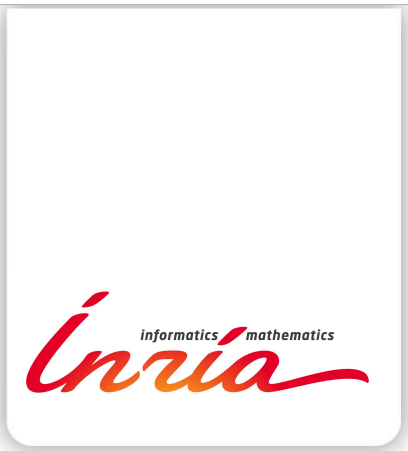
RESEARCH REPORT

N° 7838

November 2011

Project-Teams GRAAL

ISSN 0249-6399 ISRN INRIA/RR--7838--FR+ENG



Building Safe PaaS Clouds: a Survey on Security in Multitenant Software Platforms

Luis Rodero-Merino*, Luis M. Vaquero†, Eddy Caron‡, Frédéric Desprez§, Adrian Muresan¶

UMR CNRS - ENS de Lyon - INRIA - UCB Lyon 5668,
46 allée d'Italie, F-69364 Lyon, France

Universidad Politécnica de Madrid, Facultad de Informática,
B2 L3201, 28660 Boadilla del Monte, Spain

Hewlett-Packard Labs, Stoke Gufford BS34 8QZ, Bristol, UK

Project-Teams GRAAL

Research Report n° 7838 — November 2011 — 28 pages

* lrodero@fi.upm.es
† luis.vaquero@hp.com
‡ Eddy.Caron@ens-lyon.fr
§ Frederic.Desprez@ens-lyon.fr
¶ Adrian.Muresan@ens-lyon.fr

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Abstract: This paper surveys the risks brought by multitenancy in software platforms, along with the most prominent solutions proposed to address them. A multitenant platform hosts and executes software from several users (tenants). The platform must ensure that no malicious or faulty code from any tenant can interfere with the normal execution of other users' code or with the platform itself. This security requirement is specially relevant in *Platform-as-a-Service* (PaaS) clouds. PaaS clouds offer an execution environment based on some software platform. Unless PaaS systems are deemed as safe environments users will be reluctant to trust them to run any relevant application. This requires to take into account how multitenancy is handled by the software platform used as the basis of the PaaS offer. This survey focuses on two technologies that are or will be the platform-of-choice in many PaaS clouds: Java and .NET. We describe the security mechanisms they provide, study their limitations as multitenant platforms and analyze the research works that try to solve those limitations. We include in this analysis some standard container technologies (such as Enterprise Java Beans) that can be used to standardize the hosting environment of PaaS clouds. Also we include a brief discussion of Operating Systems (OSs) traditional security capacities and why OSs are unlikely to be chosen as the basis of PaaS offers. Finally, we describe some research initiatives that reinforce security by monitoring the execution of untrusted code, whose results can be of interest in multitenant systems.

Key-words: Security, Cloud, PaaS, Multitenancy, Container, Java, .NET

Étude sur la sécurité dans les plates-formes logiciels multi-utilisateurs pour la mise en oeuvre d'une infrastructure PaaS en nuage

Résumé : Ce papier étudie les risques induits par les architectures multi-utilisateurs pour les plates-formes logicielles, et les solutions les plus avancées pour résoudre ces questions. Une plate-forme multi-utilisateurs héberge et exécute des logiciels pour plusieurs utilisateurs. La plate-forme doit assurer qu'aucun code maliceux ou défaillant provenant d'un utilisateur ne vienne interférer avec l'exécution normale d'un autre code utilisateur ou avec la plate-forme en elle-même. Ce besoin de sécurité est particulièrement approprié dans les infrastructures en nuage de type PaaS (pour Platform-as-a-Service). Les PaaS en nuage offre un environnement d'exécution basé sur des plates-formes logicielles. A moins que les systèmes PaaS soient considérés comme des environnements sécurisés les utilisateurs seront récalcitrants a faire confiance à ces plates-formes pour exécuter les applications appropriées. Cela implique de prendre en compte la façon dont les multi-utilisateurs sont gérés par la plate-forme logicielle sous-jacente au PaaS. Cette étude se focalise sur deux technologies qui ont été choisis dans de nombreuses plates-formes PaaS: Java et .NET. Nous décrivons les mécanismes de sécurités qu'elles fournissent, nous étudions leurs limitations dans le cadre de plates-formes multi-utilisateurs et nous analysons les travaux de recherche qui essaye de s'attaquer à ces limitations. De plus nous évoquerons des technologies a base de containers standards (telle que Enterprise Java Beans) qui peuvent être utilisées pour standardiser l'hébergement des PaaS. De plus, nous proposons une brève discussion sur les traditionnelles niveau de sécurité dans les systèmes d'exploitation et pourquoi les systèmes d'exploitation sont peu aptes a être choisi comme base de l'offre PaaS. Enfin, nous décrivons quelques initiatives de recherche qui renforce la sécurité par le monitorage de l'exécution de code non fiable, dont les résultats peuvent être intéressant dans le cadre de systèmes multi-utilisateurs.

Mots-clés : Sécurité, Cloud, PaaS, Multi-utilisateurs, Container, Java, .NET

Contents

1 Introduction

The term *multitenancy* refers to the ability of a platform to run software from different users *in a safe manner*. To some degree, multitenancy is supported in many software platforms such as OSs or Virtual Platforms (VPs) such as Java and .NET. However, as this survey shows, none of these platforms offer a fully secured hosting environment. This problem is relevant even in controlled environments where only code from trusted users will be run: faulty code can stall its container for example by allocating too many objects (so the system runs out of memory). Security concerns are even more pressing if code from unknown users is hosted.

This work depicts how malicious code can interfere with the container platform that executes it, or with other software also hosted in the same container. Also, it presents the research works that try to solve the security limitations of standard platforms regarding multitenancy. As we will see this problem has not been neglected by the research community, but arguably it has not received as much attention as other security-related problems so far (e.g. Web attacks such as denial of service, cross-site scripting or SQL injections have been deeply studied). This is likely to change due to the growing importance of cloud systems [53] where multitenancy is specially relevant.

Cloud systems allow organizations to outsource the operation of IT infrastructure, both hardware and software. Much attention has been payed to them due to the potential benefits and business opportunities that clouds could bring [18]. However, there are several concerns that could impede the adoption

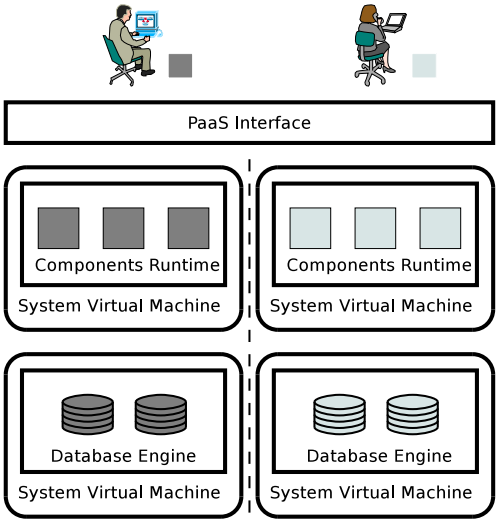
of cloud-based solutions [40]. Some of them are uncertain reliability (low availability and/or performance dropouts), vulnerability to network attacks (e.g. Denial of Service attacks), or potential vendor lock-in (users not being able to migrate their software to other clouds). Those are not addressed here as they are outside the scope of this work. Another relevant factor to be considered by potential cloud users is security: if clouds are perceived as risky environments users will be very reluctant to migrate their systems there [55]. Unfortunately, securing clouds is not a trivial task as they must face several threats. This survey focuses on the risks induced by multitenancy in *Platform-as-a-Service* (PaaS) clouds. A PaaS cloud provides a container platform where users deploy and run their components. A well known example is Google App Engine (GAE)[1], which runs Java servlets. In a PaaS cloud components from different users can be run in the same platform or container system. As we will see, this implies that malicious users have several straightforward ways to interfere with the normal execution of other components or with the container itself. This is emphasized in [52], where the authors specify that providers are responsible for isolating components so that no user software can interfere with other users. This paper further explores this requirement by surveying the isolation capabilities of potential PaaS platforms. This analysis is due at three levels representing three possible container systems: Operating System (OS) level, Virtual Platform (VP) level (i.e. Java and .NET) and container level. Most emphasis is put on the VP and container levels as, as we will see, these are more relevant for PaaS clouds.

To avoid confusion, we should clarify that there are some systems also denoted PaaS clouds that build a unique environment per user which is hosted in not shared machines, e.g. provided by an *Infrastructure-as-a-Service* (IaaS) cloud. This is the case for example of Stax.net[2] that offers pre-packaged disk images with the software stack that the user demands and where the deployment and monitoring process is eased thanks to the custom tools provided. Fig. 1(a) shows an example of such layout, where the PaaS system deploys each user's components in different Virtual Machines. In these systems it is the provider of the VMs (an IaaS provider) who is in charge of implementing proper isolation (which has its own challenges, see [48]). Hence, this paper does not deal with such PaaS systems as they delegate the implementation of secure isolation to the VM level.

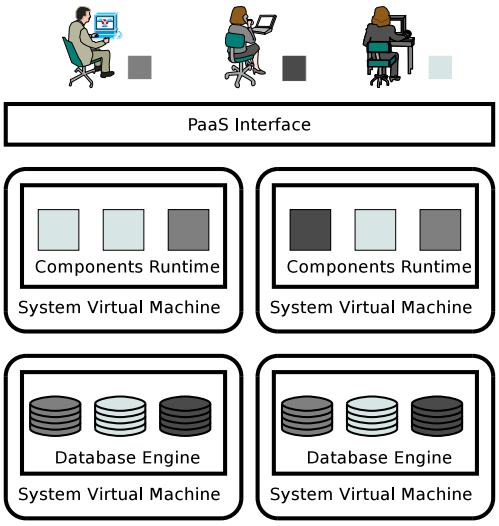
In this paper we focus instead on PaaS clouds that host and run applications from several different users in the same platform [57] in a safe manner. Tenants share PaaS platform resources (hardware, libraries, supporting services, IT management, etc.), but this is totally transparent to them. This way, the provider can host more users' applications in the same resources. Fig. 1(b) depicts such a PaaS system, where components from different users are deployed in the same container systems. To achieve safe multitenancy in PaaS platform each application must run isolated from the rest, so a malicious or faulty application cannot impact others. Also, as the code executed by the PaaS system may be untrusted, it is necessary to find mechanisms that can enforce security policies to decrease the risks involved in running such code.

The rest of this paper is organized as follows. In Section 2 we explore security

[1] `http://code.google.com/appengine`
[2] `http://stax.net`



(a) Non-shared Resources PaaS



(b) Shared Resources PaaS

Figure 1: PaaS Systems

mechanisms at OS level, while at the same time we discuss the limitations of the OS as a hosting environment for PaaS applications. Such limitations seem to signal VPs such as Java as more adequate to build PaaS systems. The most well-known container systems are based on Java, so emphasis is put on this platform. Thus, Section 3 focuses on studying standard Java features and its limitations as a PaaS container platform from the point of view of security, while Section 4 discusses security on Java container systems. Section 5 switches focus

to the .NET platform, whose security characteristics are also analyzed. With a more general approach, Section 6 comments *external code monitoring* as a security solution to be applied in PaaS platforms. Finally, Section 7 presents some conclusions resulting from this survey.

2 Safe Multitenancy through Process Isolation at Operating System Level

In [29] a complete definition of a secure Operating System is given: "*A secure OS provides security mechanisms that ensure that the system's security goals are enforced despite the threats faced by the system.*". An OS deals with resources such as devices, network, data, memory and processors. Each resource type has different security issues related with it, but to implement safe resource sharing in a multitenant OS five main security areas can be considered. 1) *Access control*, an access mechanism should be available to authorize requests from users or processes to perform OS operations as read, write, etc., on OS objects such as files, sockets, etc. The most well-known solution is based on ACL (for Access Control List), where each object has a list of permissions associated; 2) *Integrated firewall* functionality, like IP Filter, IPsec and VPN techniques; 3) *Data encryption* for data in transit or stored in the filesystem; 4) *Prevention of execution* of memory zones, using the *No execute* (Nx) page flag; 5) *Isolation* is finally the last (but not least) security area OSs must provide. Process isolation has been a basic feature of most OSs for decades. Proper isolation prevents any process to interfere with others or to access protected resources. This is achieved through well known protection mechanisms (memory segmentation and page mapping) that build a separated address space for each process. A process cannot access memory regions outside its address space. Although other ways to implement process isolation have been proposed [2], this is by far the most common in modern OSs. [39] shows how to take benefit of virtual machines to secure an OS. Also, [13] propose a Mandatory Access Control (MAC) based system to ensure integrity in OS and VMs.

Other menaces are present, but they are not so related with multitenancy (which is the main focus of this paper) or are already dealt with by the areas depicted above. Techniques like intrusion prevention, authentication and availability deal with *external attacks*; *data integrity* is preserved by ACLs and data encryption; *hidden information flows*, which occur when some user' software propagates information that should remain confidential, can happen if users share critical data (e.g. the same database), but in PaaS systems users do not share application-level data, which should prevent this kind of risks if data integrity is properly implemented.

However, typical PaaS systems do not host applications that run right on top of the OS. Although this is technically feasible, PaaS providers prefer to offer other abstractions to users. Reasons may vary:

- Platform standardization and portability. If a PaaS player allowed users to deploy applications to run on top of the OS, she/he would have to decide which OS(s) to offer, which version, and which dynamically linked libraries (and versions) should be available for applications. This is far from trivial and could constrain the set of applications that could be run

in the cloud.

- Simplified abstractions. Also, given the domain of the applications that run in PaaS clouds, providers may prefer to offer simplified abstractions that ease the development tasks.

- Dominance of interpreted languages and virtualized platforms in Web development. Finally, it is foreseeable that future developments in PaaS clouds will be strongly Web-oriented (as they will be accessed through the Web). In Web development, scripting languages (Ruby, Python and others) and virtualized platforms (Java or .NET) are dominant.

There are also several concerns that could be raised if PaaS platforms are allowed to run applications from several tenants on the same OS, [30] enumerates some of them: administration, installation, fault and attack isolation; along with crash recovery.

Furthermore, as noted in [4], general purpose OSs do not allow for an appropriate control of scheduling policies and resource management. These authors already advocated for the utilization of *containers*, although with an important difference from present container systems: those containers were an abstraction provided at the OS level. Each container encompassed the resources associated to a particular task. Each application could use one or many containers, and through them the OS was able to monitor and manage the resources (CPU, memory, bandwidth) consumed by each task executed by the application.

This same idea of 'container' is present in many systems, however they are implemented at the application level (where any resource management and tenant isolation task must be implemented).

3 Security and Multitenancy in the Java Platform

Arguably, the best well-known container systems are based on the Java platform. The Enterprise Java Bean [35] (EJB) and Servlets [34] specifications (part of the J2EE specification [36]), and the OSGi[3] specification [43] are the most relevant Java container technologies and they can be expected to have a prominent role in future PaaS platforms. For example, the GAE PaaS system already provides a runtime engine for Java servlets.

3.1 Standard Security Capabilities of Java

This section presents a brief summary of the main security features of the standard Java platform (for more information on this topic see [42, 60]). The Java specification includes the *Java security model*[4], a set of features that intend to make Java a safe environment. They include: sandbox execution so potential risks for the hosting system are limited; bytecode verification so the runtime is not corrupted; and cryptography, PKI, and secure transport APIs for communications protection. Also, Java implements a *class loading* mechanism that can

[3] The term OSGi was originally the acronym of *Open Services Gateway initiative*, but today that name is obsolete.

[4] `http://java.sun.com/javase/technologies/security/`

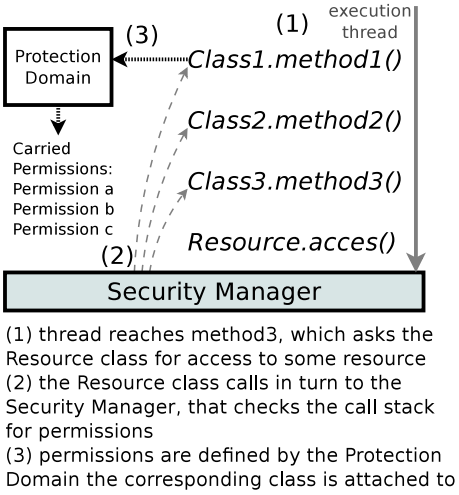


Figure 2: Checking of Permissions in Execution Stack

be used to control which classes can be instantiated by each thread. Typically, in cloud platforms untrusted code will be run by special threads with specific class loaders that limit which classes can be accessed.

Furthermore, Java implements strong *access control* capabilities to limit access to resources such as network, files, system properties, or any logical entity that the container must protect. The class loader sets for each class the *protection domain* it belongs to. This domain carries 1) a set of *permissions*; 2) the *code source*, an entity that contains the public certificates used to sign the code (if any). The security *policy*, which is set when the platform starts, is used to determine which permissions can be assigned to each class depending on its code source. Finally, the *security manager* is the entity that enforces security.

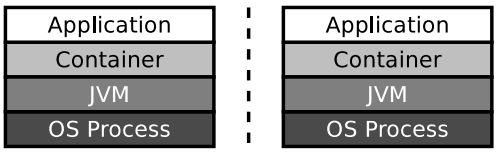
Resources are usually "wrapped" by specific classes. When some functionality needs a resource it will call the corresponding class. That class protects the resource by asking to the service manager to check if a calling thread has the corresponding permissions by traversing back the method stack. For each method in the stack, the security manager checks if the permissions carried by the protection domain the method class belongs to are enough to grant the requested access. If it finds a method in that stack belonging to a class that does not have the required permission, an exception is thrown. This is depicted in Fig. 2.

Previous control is code-centric, but can also be user-centric by using the standard *Java Authentication and Authorization Service* (JAAS) APIs. Once a user is authenticated through JAAS, one or more *principals* are associated to her. The security policy used determines which permissions are assigned to each principal when running a certain code. A more complex authorization solution (both role-based and hierarchical) oriented to multitenant clouds is presented in [10].

3.2 Security Hazards in Java

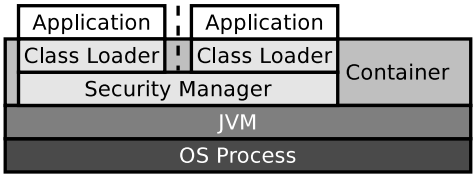
Unfortunately, the Java platform also presents certain limitations that hinder the construction of secure multitenant environments. In [27] and [6] the authors analyze the problems and threats to be taken into account when using Java as a multitenant platform. In [27] the authors also study the problems derived from running multitenant software as Java threads. As they explained, even if newer Java versions include protection mechanisms [60] so that no thread could neither modify nor stop other threads, still many issues remain:

- **Isolation**. A proper isolation mechanism must ensure that one tenant cannot access to components of other tenants. Figure 3 shows three different isolation solutions that PaaS platforms can use, ranging from isolating applications by running them on their own OS process, going through using already available security devices such as class loaders, or using last advances on virtual platforms to provide full applications isolation in the same container.



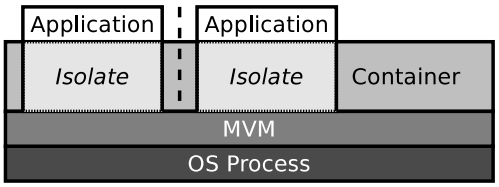
Isolation by OS processes on Virtual Platforms is expensive in terms of resource consumption.

(a) Isolation at OS Level



Isolation with standard Java security features (class loaders, access control). This does not prevent problems such as leaked references, thread termination issues, etc.

(b) Isolation by Standard Java Security



Research targetting advanced isolation abstractions provided for example by MVM.

(c) Isolation at VM Level

Figure 3: Isolation Options in PaaS Platforms

Fig.3(a) shows the most straightforward option, to create a new JVM per user application. This is a safe approach as it uses OS processes to isolate

different applications. However it is expensive in terms of resources.

Fig.3(b) depicts an approach that enforces security by means of standard Java technologies. Isolation is reinforced by class loaders. Through class loaders, a Java runtime can prevent malicious tenants from loading (and running) not allowed classes or corrupting classes used by other tenants. However, this is not enough to ensure proper isolation among tenants running code in the same JVM. Potential problems vary: visibility of object references from mutable parts of classes (specially static ones), and the possibility for malicious tenants to block other tenants through shared data structures (such as queues) or static synchronized methods [27, 15, 6, 51].

Certain research works have tried to implement the option depicted in Fig.3(c)) by providing better isolation mechanisms to Java. In [15] the authors introduce the Multitasking Virtual Machine (MVM), a modified JVM that implements the concept of *isolates*. Each isolate runs a different application (also denoted *task* by the authors) with its own threads in such a way that the application has the illusion of being executed in a non-shared VP. In MVM each task has its own memory heap and so there are no shared objects. Communication between isolates must use other mechanisms such as sockets. Depending on the amount of calls among isolates this can induce a considerable overhead. At the same time, MVM promotes the sharing of as much resources as possible to enhance performance (e.g. core native methods are shared).

Also, static variables are considered by MVM. In a typical JVM, static variables of any class are shared by all threads. In MVM, each isolate keeps its own copy of the static variables, only shared by the threads inside that isolate. Static synchronized methods in each class can be another source of trouble. The monitor associated to those methods is kept by the corresponding instance of `java.lang.Class` (in fact it is the own monitor of the instance). But in the JVM there is only one single instance of `Class` per class, shared by all threads. Hence, the monitor of the `Class` instance is also shared, so if a thread gets the monitor (by synchronizing on the `Class` instance or by calling a static synchronized method of the class) it can block any other thread trying to access it. To avoid this, MVM keeps for the same class different instances of `Class` in each isolate.

Later on, an evolution of the MVM was developed so the same MVM could support applications of different users at OS level [14]. This is implemented by controlling access to private files, allowing the safe execution of native code and adding a mechanism to ensure the correct operation of core native libraries by replicating the global state of shared core classes. Note that this work refers to users at OS level, not to be confused with the tenants of PaaS systems that will try to run their code in the VP. In a PaaS environment, it is safe to assume that the platform will always be started by a single OS level user (admin).

These and other works influenced the Java Specification Request[5] (JSR)

[5] Java Specification Requests are the standard process to define and propose new additions to the Java platform.

121 *Java Isolation API* [33], which enables Java applications to start other applications in an isolated manner. This specification defines a set of interfaces for the creation and control of isolated containers for components. However, it does not impose any implementation strategy so each isolated component could be implemented by a whole JVM running on an OS process of its own, or all isolations could share the same JVM (as in the case of MVM).

On the other hand, the JSR 121 has not been included yet in any standard release of the Java platform, and in fact it seems to be a dormant specification. Also, research project *Barcelona*[6], that hosted the development of the MVM, is no longer active[7].

KaffeOS is another interesting proposal developed by Back and Hsieh [3]. KaffeOS is a new JVM that implements support for isolated *processes* inside the runtime and manages the CPU and memory resources available to each process. These processes are similar to the ones given by typical OSs. They claim that they provide better isolation capacities that the *isolates* given by the MVM.

Geoffray et al. [22, 24] also apply the concept of *isolates* originated by the work on MVMs. However, they transform them so that they are not associated to a running task (i.e. threads can migrate among domains in contrast to isolates) but to class loaders (classes loaded by the same class loader are in the same isolation). With this approach they avoid the overhead caused by inter-task communication in the MVM. As in the case of MVM, each isolate keeps its own copy of static variables and instances of `Class`. In [24] the authors introduce *I-JVM*, a modified JVM that implements their concept of isolates. I-JVM is based on VMKit [23], a software framework to speed up the creation of VPs.

Finally, Sun et al. [51] focus on solving the problems originated by the sharing of the heap memory, such as memory leaks from faulty software that can consume all available memory. The heap is split in logical partitions, so the memory faults caused by a component only affect the partition it resides in. The partition can be repaired without rebooting the whole system.

- **Resource Accounting**. As commented before, the security manager and protection domains are the foundation of the Java environment to implement and assign custom security policies that control access to resources by code (depending, for example, on the origin of that code). Unfortunately, once access is granted to some code, that code can use the resource without limitations. There is no accounting of resource usage by threads in the Java platform, and, so, there is no way to enforce a limited utilization of resources. Therefore, a malevolent tenant can, for example, try to exhaust all available memory just by creating many instances of objects.

  The (somewhat old) *Java Virtual Machine Profiling Interface*[8] (JVMPI)

[6] `http://labs.oracle.com/projects/barcelona/`

[7] We tried to get in touch with Sun/Oracle to access to the last version of the MVM. We were notified that, although there is a more recent and stable version based on JDK 7, access to the MVM has been restricted since Oracle acquired Sun.

[8] `http://java.sun.com/j2se/1.4.2/docs/guide/jvmpi/jvmpi.html`

and its more recent replacement the *Java Virtual Machine Tooling Interface*[9] (JVMTI) can be used to support resource accounting as they allow to inspect the state of applications and the JVM. However, these interfaces must be used by software written in native code, breaking Java portability. Also, they introduce a considerable overhead that can make them unusable in many production environments. Finally, these interfaces do not aim at accounting of generic resources.

There have been several approaches trying to solve this for different single resources. For example [47] proposes a system able to account memory usage by using a modified garbage collector that computes the total size of objects reachable by each task as it looks for unreachable objects. They are deemed to be imprecise due to shared references [3].

Other works apply bytecode rewriting (also called program transformations) to inject some kind of accounting capabilities to the Java platform in a portable way. This manner, the platform should be able to prevent threads from using too many resources. The most prominent efforts using this approach are JRes [16] and JRAF-2 [8, 9, 28, 7].

As a result of this concern about the lack of a proper resource control mechanism in Java, Czajkowski and others started to work in a new Resource Management (RM) API [17]. This work and the MVM [15] (discussed above) are strongly related. The RM uses MVM's idea of *isolates* as the basic accounting entity that can demand or dispense resources, and [17] introduces an implementation of the RM API on top of the MVM.

Eventually [17] leaded to the creation of the JSR 284 *Resource Consumption Management API* [37]. This JSR, which has been recently approved, defines a set of interfaces that enable the programming of resource management policies. This API *"will be a framework through which resources can be uniformly exposed to client programs as entities subject to management"*. Also, JSR 284 includes a set of core resources that all compliant implementations will have to expose by default. An implementation is already available, but it is unknown if this API will be included in future releases of any of the flavors (J2ME, J2SE, J2EE) of the standard Java platform.

On the other hand, KaffeOS implements per process resource accounting and bounds setting (CPU and memory). It does not provide accounting of other resources neither from the platform nor handled by the users.

Regarding I-JVM, it implements per-isolate accounting of CPU time, threads created, I/O r/w operations and memory. But as KaffeOS, it does not have a general accounting framework for a generic resource.

- **Safe Thread Termination**. This problem is due to the lack of a safe way to enforce the termination of a Java thread. The `java.lang.Thread.stop()` method was intended for that, but:
  - It is *deprecated* because it is deemed unsafe: the terminated thread would release all its monitors, which could leave some objects in an inconsistent state.

[9] `http://java.sun.com/javase/6/docs/technotes/guides/jvmti`

– The method triggers a `java.lang.ThreadDeath` exception in the thread to stop it. The thread can just catch that exception and ignore it to keep running.

Hence malicious threads can remain alive forever, consuming resources trying to monopolize resources, block other threads, etc. Another problem could be caused by the platform trying to run a safe shutdown, which implies that all threads running inside the platform must be stopped first. If the platform waits for a malicious thread to terminate then it could be brought to a stall state. Some solutions [49] propose to modify the untrusted software bytecode to inject termination checks at certain execution points. These solutions have a drawback: they incur heavy performance penalties.

MVM does solve this problem. A MVM-aware application can create, execute, pause, resume and stop other applications. Also, KaffeOS allows to stop the *processes* it is based on.

Finally, in I-JVM, when one isolate is terminated all the threads originated by it are stopped by a special `StoppedIsolateException` exception that can only be caught by objects outside the terminated isolate (so the exception cannot be ignored by the isolate being stopped).

But I-JVM, on the other hand, does not totally implement safe thread termination. The problem is that in I-JVM the same thread can traverse different domains regardless its origin (this cannot happen in MVM nor in KaffeOS) as isolations are not based on threads unlike MVM isolates or KaffeOS processes. When one thread is stopped all the monitors locked by it are released, which could leave objects synchronized by those locks in an inconsistent state. In I-JVM this could happen when releasing the locks of objects in isolates other than the one being stopped. This is the same reason because the standard `java.Thread.stop()` method was deprecated in Java. The creators of I-JVM estimate that the benefits of light inter-isolation communication outweigh this problem.

4 Security in Java Application Containers

It is to be expected that future PaaS clouds will not run user components right on top of the JVM. It seems more likely they will use container technologies to provide added standard services. In [26], the authors identify the security threats that multitenant containers must address and enumerate the security requirements they must fulfill:

- *Availability*: an application shall not use local or connected resources that prevent other applications from running due to resource starvation. The container should have mechanisms to enforce different resource sharing policies. Also, the container must be available regardless of the state of the applications running inside.
- *Confidentiality and Integrity*: an application shall not explore or modify the platform of other applications if not authorized. Access to other applications and their data must be controlled.

It is straightforward to see that these requirements would be achieved by properly addressing the issues listed in Section 3. Container availability can be brought by safe thread termination and resource accounting, while confidentiality and integrity would be implemented by full isolation of components.

The remainder of this section focuses on the security features of J2EE and OSGi technologies, as they are the most prominent relevant Java container solutions today. Also, the works that try to bring stronger security capabilities to each container technology are listed.

4.1 J2EE Containers

The EJB specification [35], as part of the contract between the EJB and the container, imposes strong restrictions and limitations to what EJBs can do. EJBs cannot create threads (to avoid interferences with the container's ability to control components' lifecycle), manipulate files (files are not transactional resources and could also limit the application distributability), modify class loaders, access non final static fields (such fields would make a bean difficult to distribute), etc.

These restrictions are enforced by the EJB container through the standard Java security model (see Section 3.1), and all together build an interesting security mechanism. EJB containers combine these constraints with the application of class loaders to achieve proper EJBs isolation. Unfortunately, these restrictions impose a somewhat limited programming model which may not be appropriate for many development needs. And, more important, they are not enough to fully achieve the requirements listed in Section 3.2.

On the other hand, the Servlet specification, which is also part of the J2EE platform (as the EJB specification), does not stress isolation among servlets, nor imposes strict restrictions for servlet programming. In this specification, security is concerned only with authentication and authorization of servlets' clients.

It is possible, of course, to apply the standard security Java mechanisms (such as access control and PKI APIs) to the development of servlets and EJBs based systems. There are texts available that address this topic [61, 62]. But even in this case, proper *Isolation*, *Resource Accounting* and *Safe Thread Termination* (Section 3.2) would remain as open issues.

Some research works [32, 31] have tried to use MVM (see Section 3.2) to achieve proper isolation among users applications on J2EE environments. In [32] the authors discuss how to apply MVM's isolates in a J2EE server. They propose using *application domain* isolates, where one application domain encapsulates one or more user J2EE applications, including its required servers. Later on, in [31] the authors used a MVM extended with the Resource Management API (defined in [17], see Section 3.2) and combine it with application domain isolates, so they can easily monitor the resources used by each application.

4.1.1 A Servlets-Based PaaS: Google App Engine

Being a prominent PaaS platform, based totally in the Java Virtual Platform, it is worth to discuss how GAE has addressed the security problems of standard Java.

First, they limit the possible actions that users can perform applying the Java security model, i.e. they apply custom class loaders and security policies enforced by the Security Manager. For example, tenants cannot create new threads, instantiate certain classes, modify system properties or read files that do not belong to the user application (a GAE application is basically a set of Java servlets, Javascript code, configuration files and static content like images or HTML pages).

Regarding isolation, GAE solves it in a quite "naive" manner: users do not share servers. Each user application runs on its own JVM instance (as depicted in Fig. 3(a)).

GAE offers accounting data of certain resources: CPU, network bandwidth and stored data size. Users are billed depending on the amount of resources used. However, it is not explained how GAE performs this accounting (using a custom JVM, using the JVM Tooling Interface, at OS level, etc.).

Finally, GAE uses thread termination to control how long it takes to attend each request. A request in GAE can last up to 30 seconds. When that limit expires, an exception is thrown by the platform to the servlet processing the request. If the exception is not caught, the thread will finish and a `HTTP 500 server error` message will be sent in response to the HTTP request that triggered the thread execution. If the exception is caught the runtime engine will give *"the request handler a little bit more time (less than a second) after raising the exception to prepare a custom response"*. After that, the thread is terminated by force. Google claims that the thread is shutdown "gracefully", other threads in the same server are not affected. In fact, the whole container is stopped. To make sure that other threads are not affected, the load balancer in front of the container stops sending requests to it when a thread is to be stopped. Then, when no more threads are running, the whole container server can be stopped. This implies that programmers should develop servlets taking into account that requests should be attended by stateless processes (there is no concept of session affinity per user) as consecutive requests from the same user can be forwarded to different server instances.

4.2 OSGi Containers

The OSGi framework defines a platform where loosely coupled software modules (denoted *bundles*) can expose and use services; OSGi enforces some isolation through its *Module Layer*. This layer defines a modularization model so that packages included in each bundle are shared (exported) or hidden to other bundles as declared by the developer. Again, this isolation is implemented through class loaders.

Extra security is provided by controlling whether bundles can export/import certain packages, access resources, etc. Still, OSGi carries several potential security hazards. In [45] the authors enumerate 25 different security flaws in different OSGi implementations. And, while 17 of them can be fixed programmatically by setting proper security measures, there are still 8 vulnerabilities that need to be addressed at JVM level. All of them are related with the security limitations of Java mentioned in Section 3.2: poor isolation (e.g. a bundle can modify shared static variables), need for resource accounting (e.g. a bundle could use all of the memory available) and lack of support for thread termination (e.g. a bundle can ignore signals to stop and catch all `ThreadDeath` exceptions).

Some works try to improve the OSGi framework robustness by providing better isolation: Gama and Donsez [21] patch an OSGi implementation using the Isolation API (JSR 121) on MVM to provide service level isolation.

In [24] the authors modify an OSGi implementation to run with I-JVM. They show how applying I-JVM this new OSGi platform solves the 8 risks described in [45] tied to the JVM.

Other works try to enhance the tolerance to faulty software, for example in [1] the authors use light proxies to route calls between bundles that wrap service objects and handle failures when they occur.

5 Security Considerations about the .NET Platform as a PaaS Enabler Technology

No any other VP has been as intensively studied as the Java platform. Also, no other VP has reached the same popularity. But Java is not the only candidate VP that can be used to build a PaaS system. This section will introduce the main security features of the .NET platform, which can be regarded as an alternative to the Java platform.

5.1 Standard Security Capabilities of .NET

The .NET platform is a development environment created by Microsoft with several similarities with the Java platform. The *Common Language Runtime* (CLR), which would be the equivalent to the JVM in .NET settings, implements the main security aspects of this platform.

In .NET, software is contained in libraries denoted *assemblies*, which are grouped in *code groups*. Membership of code groups is ruled by the *evidences* that each assembly carries (for example who signs the code). Each code group has an associated set of *permissions*. If some assembly belongs to more than one group, its associated permissions are the union of all the permissions of all groups it belongs to.

The mapping between code groups and permissions is done through *security policies*. Policies are organized in a hierarchy with 4 levels (top-down order): enterprise, machine, user and application domain. Usually, the permission associated to each code group is given by the intersection of the permissions at all levels it belongs to, although more complex settings are possible.

Permissions are used for granting access to resources or to other code. They have a stack walking semantics very similar to the one found in Java. If a method demands a certain permission, then all the methods higher than the current one in the call stack are checked for that permission. This prevents attacks in which some untrusted software tries to use a trusted piece of code to run a protected operation.

We can see that the CLR access control mechanism has similarities with the one used in Java, although it is considered by some [46] as easier to use.

5.2 Security Hazards in .NET

- **Isolation**. The CLR implements the concept of *Application Domains* (ADs). Each application is assigned an AD when is run by the CLR (the

same CLR instance can run several ADs with several instances of different applications). ADs are isolated, so code running in one AD neither can call, nor can be called from code running in other AD. If several application instances use the same code, the CLR will handle one copy of that code per AD where it is used. For intra-process isolation in .NET, using different application domains is recommended because they can be dynamically loaded and unloaded during the runtime of the application.

An interesting feature of the CLR is that it keeps a separate copy of the static variables maintained for each domain, thus preventing object references from being leaked across domains as static variables. We can conclude that, by default, the CLR has more complete (and thus safer) isolation capabilities than the standard JVM. However, although the *application domain* concept provides a straightforward way to achieve tenancy isolation, the fact is that CLR still suffers some other limitations of the Java platform.

- **Resource Accounting** Just like in the case of Java, .NET does not implement any generic resource accounting functionality. It does have a profiling mechanism, but it provides information about the state of the CLR through events (load/unload of classes, threads creation, and others), it cannot be used by components developers to control the resources they offer.

  There has been some works around resource accounting that target Windows applications. Notably [44] have described a framework that allows resource accounting. This framework allows the dynamic assignment of resources to tasks and task management to a fine granularity that includes bounding the running context of tasks (for example in CPU and memory usage) therefore creating a sandboxed context for the task. The framework described here targets *unmanaged code* (code that does not target the .NET framework and is not run by the CLR) but the authors stated it was being extended to allow .NET remote resources to be used. As such the presented framework is a viable solution for resource accounting for the .NET framework.

- **Safe Thread Termination**

  CLR's thread termination solution is based on a C#'s method (`System.Threading.Thread.Abort()`) that injects an exception in the aborted thread, as the `java.lang.Thread.stop()` does in Java. The `Abort()` method is not deprecated (as the `stop()` method is), yet it is recommended to avoid it[10]. But even more important is the fact that .NET does not guarantee that the thread on which `Abort()` was called is stopped. In fact it is easy for the thread to continue its execution by handling the exception and calling `System.Threading.Thread.ResetAbort()` or by having unbound computations in its `catch` or `finally` statements. Thus, like Java, .NET does not provide a safe mechanism for thread termination.

  This impacts ADs management. Before unloading an application domain all its threads must be stopped, which is implemented by using

[10] http://msdn.microsoft.com/en-us/library/system.threading.thread.abort.aspx

the `Thread.Abort()` method. Note that, given the fact that thread stopping is not guaranteed neither is the successful unloading of an application domain.

5.3 Security in .NET Application Containers

Regarding container architectures, no container system similar to Java's EJBs or OSGi exists in .NET. The closest technologies could be *ASP.net*[11] and *Component Object Model*[12] (COM). ASP.net provides a Web framework, but as in the case of the Servlets specification there is no special reinforcement of isolation among components (although it uses the concept of ADs). Regarding the COM platform, it is not built on top of .NET and is not part of the .NET framework. Also, COM is not a container technology *per se*, it is more oriented to enabling the connection of components. COM+ has been developed as an improvement of COM. Recent versions of COM+ add private / public component isolation mechanisms whereas previous versions only offered role-based authorization. For its use in the .NET framework, a wrapper library has been built under the name of .NET Enterprise Services[13].

The compliance and possible implementation of an OSGi-like platform on the .NET framework has been studied by [20]. To enforce OSGi-like containers in .NET, the authors recommend applying ADs. They can provide the necessary isolation mechanisms, yet the only way to communicate between two non-shared application domains is by using interprocess communication solutions such as .NET remoting. These communication mechanisms come with a considerable time overhead which would make some applications impractical, yet the possibility of an OSGi-like platform implemented on top of the .NET framework exists.

There have been a few projects that aim towards the development of a PaaS cloud based on the .NET framework. One such project is the Aneka Cloud Platform described in [54]. The goal of the Aneka project is to provide a PaaS cloud that enables the deployment of public, private or hybrid clouds. The Aneka platform is based on Aneka containers. They provide the services required for platform management and the runtime necessary for the execution of applications.

Security inside the Aneka platform is handled by providers of authentication and authorization. The providers have the role of abstracting the concrete mechanisms that perform the task. As such, Aneka is able to use the underlying authentication and authorization mechanisms of the environment in which it was deployed if required and also to provide custom ones.

Although the general mechanisms used for application isolation in current cloud environments have been presented, the specifics implemented in Aneka related to this domain have not been detailed in the referenced work. As a result, the reader is unsure if Aneka contains implicit isolation or sandboxing for its deployed applications or if the Aneka user is responsible for developing her/his own isolation mechanisms.

In a previous work [12] Aneka has been described as an enterprise grid platform. In addition to the membership-based security approach described above,

[11] http://www.asp.net
[12] http://www.microsoft.com/com
[13] http://msdn.microsoft.com/en-us/library/Aa286569

[12] also presents the possibility of using a certification-based approach with X.509 certificates for authentication. No further details related to the application isolation mechanisms used are given.

6 Monitoring External Code Execution to Enforce Security

The security features of VPs and the related research efforts studied so far try to build a safe environment by addressing the platform characteristics that can be used by malevolent code (e.g. not proper thread termination mechanisms).

Another complementary approach is to *monitor untrusted code execution* to ensure that security policies are fulfilled by tenants' code. For example a security policy that could be enforced in PaaS systems is to impose tenants code to apply SSL connections when sockets are used. Relevant research works related with this approach are analyzed in this section.

The components that monitor code execution and take actions when some policy is violated are denoted *Reference Monitors*. Schneider in [50] presents 1) a formalism to determine which security policies can be reinforced by what he denotes *Execution Monitoring* (EM); 2) an automata-based mechanism to define such policies. The formalism uses a set of restrictions: EM only uses the information obtained by observing the code execution, it does not modify the code observed. It truncates the code execution when some security policy is violated.

On the other hand, although Schneider's definition of EM does not include any mechanism that modifies the executed code, such solutions are also considered by other authors as EM. Schneider himself states that nothing prevents using such approach with arbitrary security automata [50].

Security monitors that modify the untrusted code are denoted *Inline Reference Monitors* (IRM). Some examples of IRM based solutions are

- SASI [59], it adds code that 1) simulates an automaton that enforces a certain security policy and 2) it is executed before each untrusted code instruction.
- Java-MaC [38], an implementation in Java of the Monitoring and Checking architecture, which ensures that the code runs correctly with regards to a formal specification of requirements.
- Polymer [5], it allows to define monitors in the Polymer language and translates them to Java bytecode, which is then used to rewrite the untrusted code.

The idea of weaving security enforcement code inside untrusted modules is clearly related with *Aspect Oriented Programming* (AOP). AOP [19] intends to provide mechanisms to define "crosscutting concerns", or aspects, that are present in different components of the same system. Security is one of such concerns, as many components (if not all) must take into account security policies and constraints.

Through AOP a PaaS platform could reinforce security rules in a transparent manner [56], like for example log relevant data, implement protection

techniques against buffer overflows, etc. The Polymer system is in fact using an approach similar to AOP. Java-MOP [11] also applies AOP to monitor formal specifications in programs. In a recent work [25] the authors present an XML-based language to express security rules as automata whose edge labels (i.e. transitions) become AOP *pointcuts*, that is, places in the code affected by a certain aspect and where the IRMs will be injected. A more straightforward application of AOP to security is found in [58]. Here the authors apply AOP to add role-based access control to a CORBA access control system. Also, users could apply AOP to point out in which parts of the service some security policies must be checked.

Rather than injecting extra code to untrusted applications, other solutions are oriented to the static analysis of software before execution to ensure that it does not break any security police. For example, *Proof Carrying Code* [41] (PCC) carries static information that can be examined before execution to prove that the code is safe. It is unlikely however that in PaaS systems such extra information will be available.

Feature	**JVM**	**CLR**	**MVM**	**I-JVM**	**KaffeOS**
Access control mechanisms	Based on Permissions and Policies	Based on Permissions and Policies	Similar to JVM	Similar to JVM	Similar to JVM
Reference leak	Not fixed	Fixed with ADs	Fixed with Isolations	Fixed with Isolations	Fixed with Processes
Shared static references	Not fixed	Fixed with ADs	Fixed with Isolations	Fixed with Isolations	Fixed with Processes
Block by synchronized static components	Not fixed	Fixed with ADs	Fixed with Isolations	Fixed with Isolations	Fixed with Processes
Thread termination	Not fixed	Not fixed	Fixed with Isolations	Not Fixed	Fixed with Processes
Resource accounting	Profiling through JVMTI. Resource accounting specified by JSR 284	Profiling mechanism	Generic resource management API	CPU, memory, #threads, I/O	CPU and memory

Table 1: Summary of Security Features of Virtual Platforms

7 Discussion and Conclusion

As cloud adoption grows, also there will be an increasing demand for multitenant platforms that allow to run, in a safe manner, untrusted code from different users in the same container system.

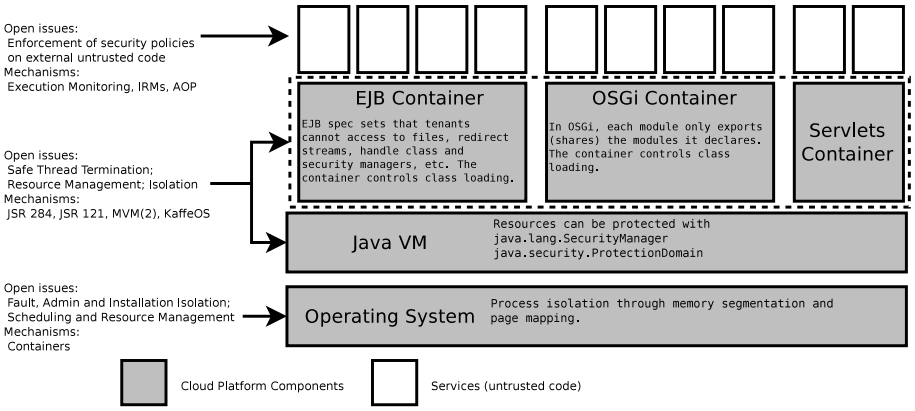


Figure 4: Summary of PaaS Security Issues and Solutions at Different Layers.

But present standard VPs, that could be used as the basic building blocks of PaaS clouds, still suffer from some important security flaws that must be taken into account when designing a PaaS system. Figure 4 summarizes the main open security issues at each level of a Java PaaS platform. Also, for each level the figure briefly enumerates both the solutions presented in this survey to address those issues, along with the security mechanisms already implemented.

Table 1 summarizes the security features discussed in this paper for different VPs. The *access control mechanism* security feature in that table refers to the standard security mechanisms explained in Sections 3.1 and 5.1. The *reference leak*, *shared static references*, *block by synchronized static components*, *thread termination* and *resource accounting* features are discussed in Sections 3.2 (for the JVM, MVM, I-JVM and KaffeOS VPs) and 5.2 (for the CLR VP). From the analysis carried out in those sections it can be concluded that the standard Java platform still has some limitations that hinder the safe execution of untrusted code, a capability that we deem necessary for the construction of PaaS systems. The CLR on the other hand implements more powerful isolation characteristics that solve some of the problems present in Java. However it seems that Java is better positioned as a base platform for building PaaS clouds. First, the CLR still lacks a safe mechanism for thread termination and a generic resource accounting framework (which is addressed in Java by JSR 284). Also, remarkable container technologies are based on the Java platform (J2EE and OSGi) and it is reasonable to expect them to be the basis of several PaaS platforms (as they are already). Furthermore, much research effort has been put on the JVM to address its security limitations (MVM, KaffeOS, I-JVM). Of all these works, MVM seems the more complete solution as it answers all open security issues. I-JVM, on the other hand, takes a different approach to isolation, so they allow threads to traverse different isolates. This way they solve the high costs of inter-isolate communication present in MVM and KaffeOS. However, due precisely to its design, I-JVM does not solve the thread termination issue. Designers of secure PaaS systems should decide which approach better suits their needs.

Besides the security guarantees achieved by the platform, security in PaaS clouds must address other aspects. First they must try to enforce security poli-

cies so users do not build applications that are themselves prone to attack. This can be done through the enforcement of security policies by the code monitoring techniques studied above. A survey of research in this area shows that most proposals are based on AOP in the Java platform, which further positions Java as a good candidate to build secure PaaS clouds.

In any case, future work on VPs and container systems (which will impact on the security of PaaS clouds) should take into account the risks brought by multitenancy outlined in this work. They should use or develop artifacts that bring full isolation among components, blocking access to external references. Also, it must be possible to stop non-trusted threads without affecting the platform, and mechanisms that allow to implements resource sharing policies should be available.

Acknowledgements

This research has been partially funded by the EC under project CumuloNimbo FP7-257993, by the Madrid Research Council (CAM) under project CLOUDS S2009TIC-1692 and by the Spanish Science Foundation under project CloudStorm TIN2010-19077.

References

[1] Ahn, H., Oh, H., Sung, C. O., April 2006. Towards reliable OSGi framework and applications. In: Proceedings of the 2006 21st ACM Symposium on Applied Computing (SAC'06), pp. 1456–1461.

[2] Aiken, M., Fähndrich, M., Hawblitzel, C., Hunt, G., Larus, J., October 2006. Deconstructing process isolation. In: Proceedings of the 2006 Workshop on Memory system performance and correctness (MSPC'06), pp. 1–10.

[3] Back, G., Hsieh, W. C., July 2005. The KaffeOS Java runtime system. ACM Transactions on Programming Languages and Systems (TOPLAS) 27, pp. 583–630.

[4] Banga, G., Druschel, P., Mogul, J. C., February 1999. Resource containers: A new facility for resource management in server systems. In: Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI'99). pp. 45–58.

[5] Bauer, L., Ligatti, J., Walker, D., June 2005. Composing security policies with polymer. ACM SIGPLAN Notices 4, pp. 305–314.

[6] Binder, W., April 2006. Secure and reliable Java-based middleware - challenges and solutions. In: Proceedings of the First International Conference on Availability, Reliability and Security 2006 (ARES'06), pp. 662–669

[7] Binder, W., Hulaas, J., October 2006. Exact and portable profiling for the JVM using bytecode instruction counting. Electronic Notes in Theoretical Computer Science 164, pp. 45–64.

[8] Binder, W., Hulaas, J. G., October 2001. Portable resource control in Java - the J-SEAL2 approach. In: Proceedings of the 2001 16th ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA'01), pp. 139–155.

[9] Calderón, V., Binder, W., June 2002. JRAF–the Java resource accounting facility. In: Proceedings of the 2002 1st Workshop on Resource Management for Safe Languages (ECOOP'02).

[10] Calero, J. M. A., Edwards, N., Kirschnick, J., Wilcock, L., Wray, M., December 2010. Toward a multi-tenancy authorization system for cloud services. IEEE Security and Privacy 8, pp. 48–55.

[11] Chen, F., Grigore, R., April 2005. Java-MOP : A monitoring oriented programming environment for Java. In: Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'05). Vol. 3440 of Lecture Notes in Computer Science, Springer, pp. 546–550.

[12] Chu, X., Nadiminti, K., Jin, C., Venugopal, S., Buyya, R., December 2007. Aneka: Next-generation enterprise grid platform for e-science and e-business. In: Proceedings of the 3rd IEEE International Conference on e-Science and Grid Computing (eScience'07), pp. 151–159.

[13] Clemente, P. and Rouzaud-Cornabas, J. and Toinard, C. 2010. From a Generic Framework for Expressing Integrity Properties to a Dynamic MAC Enforcement for Operating Systems. Transactions on Computational Science XI. Springer. pp. 131–161.

[14] Czajkowski, G., Daynès, L., Titzer, B., June 2003. A multi-user virtual machine. In: Proceedings of the 2003 USENIX Annual Technical Conference. USENIX Association, p. 7.

[15] Czajkowski, G., Daynés, L., November 2001. Multitasking without compromise: a virtual machine evolution. ACM SIGPLAN Notices 36, pp. 125–138.

[16] Czajkowski, G., Eicken, T. V., October 1998. JRes: A resource accounting interface for Java. In: Proceedings of the 1998 13th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'98), pp. 21–35.

[17] Czajkowski, G., Hahn, S., Skinner, G., Soper, P., Bryce, C., November 2004. A resource management interface for the Java platform. Software: Practice and Experience 35, pp. 123–157.

[18] Economist, T., October 2009. Clash of the clouds. The Economist, Online version available.
http://www.economist.com/displaystory.cfm?story_id=1463%7206

[19] Elrad, T., Filman, R. E., Bader, A., October 2001. Aspect-oriented programming: Introduction. Communications of the ACM 44, pp. 29–32.

[20] Escoffier, C., Donsez, D., Hall, R. S., January 2006. Developing an OSGi-like service platform for .NET. In: Proceedings of the 3rd IEEE Consumer Communications and Networking Conference (CCNC'06). Vol. 1, pp. 213–217.
http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1%593018

[21] Gama, K., Donsez, D., June 2009. Towards dynamic component isolation in a service oriented platform. In: Proceedings of the 2009 12th International Symposium on Component-Based Software Engineering (CBSE'09). Vol. 5582 of Lecture Notes in Computer Science, Springer, pp. 104–120.

[22] Geoffray, N., Thomas, G., Clément, C., Folliot, B., April 2008. Towards a new isolation abstraction for OSGi. In: Proceedings of the 2008 1st Workshop on Isolation and integration in embedded systems (IIES'08), pp. 41–45.

[23] Geoffray, N., Thomas, G., Lawall, J., Muller, G., Folliot, B., March 2010. VMKit: a substrate for managed runtime environments. In: Proceedings of the 6th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments (VEE'10), pp. 51–62.

[24] Geoffray, N., Thomas, G., Muller, G., Parrend, P., Frénot, S., Folliot, B., 2009. I-JVM: a Java Virtual Machine for component isolation in OSGi. Research Report RR-6801, INRIA.
`http://hal.inria.fr/inria-00354580/en/`

[25] Hamlen, K. W., Jones, M., June 2008. Aspect-oriented in-lined reference monitors. In: Proceedings of the 3rd ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS'08), pp. 11–20.

[26] Herzog, A., Shahmehri, N., June 2005. An evaluation of Java application containers according to security requirements. In: Proceedings of the 2005 14th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprise (WETICE'05), pp. 178–183.

[27] Herzog, A., Shahmehri, N., September 2005. Problems Running Untrusted Services as Java Threads. Vol. 177/2005 of IFIP International Federation for Information Processing, Springer, pp. 19–32.

[28] Hulaas, J., Binder, W., August 2004. Program transformations for portable CPU accounting and control in Java. In: Proceedings of the 2004 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation (PEPM'04), pp. 169–177.

[29] Jaeger, T., 2008. Operating System Security. Synthesis Lectures on Information Security, Privacy, and Trust. Morgan & Claypool Publishers.

[30] Jiang, X., Xu, D., 2003. Protection of application service hosting platforms: an operating system perspective. Position paper, Purdue University.
`http://www.cs.purdue.edu/homes/dxu/pubs/SODA-protection%.pdf`

[31] Jordan, M., Czajkowski, G., Kouklinski, K., Skinner, G., October 2004. Extending a J2EE server with dynamic and flexible resource management. In: Proceedings of the 2004 5th ACM/IFIP/USENIX international conference on Middleware. Vol. 78 of Middleware Conference, Springer, pp. 439–458.

[32] Jordan, M., Daynès, L., Jarzab, M., Bryce, C., Czajkowski, G., June 2004. Scaling J2EE application servers with the multi-tasking virtual machine. Tech. Rep. TR-2004-135, SUN Microsystems.

[33] JSR121, June 2006. Java Specification Request 121: Java Isolation API. Available Online.
http://jcp.org/en/jsr/detail?id=121

[34] JSR154, September 2007. Java Specification Request 154: Java Servlet 2.5 Specification. Available Online.
http://jcp.org/en/jsr/detail?id=154

[35] JSR220, May 2006. Java Specification Request 220: Enterprise Java Beans 3.0. Available Online.
http://jcp.org/en/jsr/detail?id=220

[36] JSR244, May 2006. Java Specification Request 244: Java Platform, Enterprise Edition 5 (Java EE Specification). Available Online.
http://jcp.org/en/jsr/detail?id=244

[37] JSR284, January 2009. Java Specification Request 284: Resource Consumption Management API. Available Online.
http://jcp.org/en/jsr/detail?id=284

[38] Kim, M., Viswanathan, M., Kannan, S., Lee, I., Sokolsky, O., October 2004. Java-MaC: A run-time assurance approach for java programs. Formal Methods in System Design, pp. 129–155.

[39] King, S. T., Dunlap, G. W., Chen, P. M., 2003. Operating system support for virtual machines. In: USENIX Annual Technical Conference, General Track. USENIX, pp. 71–84.

[40] Leavitt, N., January 2009. Is cloud computing really ready for prime time? Computer 42, pp. 15–20.

[41] Necula, G. C., Lee, P., June 1998. The design and implementation of a certifying compiler. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (SIGPLAN'98), pp. 333–344.

[42] Oaks, S., May 2001. Java Security. O'Reilly.

[43] OSGi, June 2009. OSGi Service Platform Core Specification. Available Online.
http://www.osgi.org/Release4/HomePage

[44] Parlavantzas, N., Coulson, G., Blair, G., June 2003. A resource adaptation framework for reflective middleware. In: Proceedings of the 2nd International Workshop on Reflective and Adaptive Middleware (located with ACM/IFIP/USENIX Middleware'03).

[45] Parrend, P., Frenot, S., April 2009. Security benchmarks of OSGi platforms: toward hardened OSGi. Software- Practice & Experience 39, pp. 471–499.

[46] Piliptchouk, D., December 2008. Java vs. .NET Security. O'Reilly.

[47] Price, D. W., Rudys, A., Wallach, D. S., May 2003. Garbage collector memory accounting in language-based systems. In: Proceedings of the 2003 IEEE Symposium on Security and Privacy (SP'03), pp. 263–274.

[48] Ristenpart, T., Tromer†, E., Shacham, H., Savage, S., November 2009. Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. In: Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS'09), pp. 199–212.

[49] Rudys, A., Wallach, D. S., May 2002. Termination in language-based systems. ACM Transactions on Information and System Security 5, pp. 138–168.

[50] Schneider, F. B., February 2000. Enforceable security policies. ACM Transactions on Information and System Security 3 (1), pp. 30–50.

[51] Sun, K., Li, Y., Hogstrom, M., Chen, Y., October 2006. Sizing multispace in heap for application isolation. In: Proceedings of the 2006 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications (OOPSLA'06), pp. 647–648.

[52] Takabi, H., Joshi, J. B. D., Ahn, G.-J., December 2010. Security and privacy challenges in cloud computing environments. IEEE Security and Privacy 8, pp. 24–31.

[53] Vaquero, L., Rodero-Merino, L., Caceres, J., Lindner, M., 2009. A break in the clouds: Towards a cloud definition. ACM Computer Communication Review 39 (1), pp. 50–55.

[54] Vecchiola, C., Chu, X., Buyya, R., January 2010. Aneka: A Software Platform for .NET-based Cloud Computing. Vol. 18 of Advances in Parallel Computing. IOS Press, pp. 267–298.

[55] Viega, J., Aug 2009. Cloud computing and the common man. Computer, 106–108.

[56] Viega, J., Bloch, J., Chandra, P., February 2001. Applying aspect-oriented programming to security. Cutter IT 14, pp. 31–39.

[57] Weissman, C. D., Bobrowski, S., June 2009. The design of the force.com multitenant internet application development platform. In: Proceedings of the 5th SIGMOD International Conference on Management of Data (SIGMOD'09), pp. 889–896.

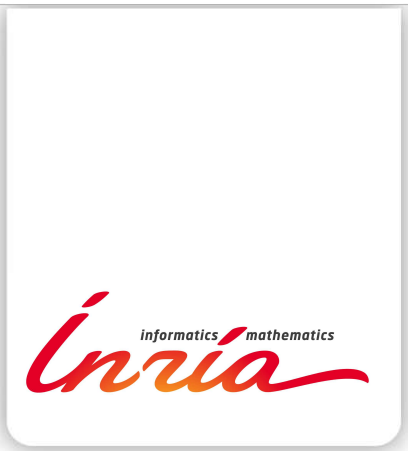
[58] Yi, S. G., Deng, Y., Yu, H., He, X., Beznosov, K., Cooper, K., June 2004. Applying aspect-orientation in designing security systems: A case study. In: Proceedings of the 16th International Conference of Software Engineering and Knowledge Engineering (SEKE'04), pp. 360–365.

[59] Úlfar Erlingsson, Schneider, F. B., September 1999. SASI enforcement of security policies: a retrospective. In: Proceedings of the Workshop on New Security Paradigms (NSPW'99), pp. 87–95.

[60] Gong, L., Ellison, G., Dageforde, M., June 2003. Inside Java 2 Platform Security: Architecture, API Design and Implementation. Prentice Hall.

[61] Kumar, P., September 2003. J2EE Security for Servlets, EJBs, and Web Services. Prentice Hall.

[62] Pistoia, M., Nagaratnam, N., Koved, L., Nadalin, A., February 2004. Enterprise JavaTM Security: Building Secure J2EETM Applications. Addison Wesley.



RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399